2

RSRE
MEMORANDUM No. 4367

ROYAL SIGNALS & RADAR ESTABLISHMENT

SYNTAX AND LEXIS OF THE SPECIFICATION LANGUAGE Z

Author: C T Sennett

PROCUREMENT EXECUTIVE,
MINISTRY OF DEFENCE,
R S R E MALVERN,
WORCS.

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Memorandum 4367

Title: Syntax and lexis of the specification language Z

Author: C T Sennett

Date: February 1990

ABSTRACT

This memorandum is presented as a contribution to the standardization of the specification language Z. It deals with the presentation issues which need to be settled in the definition of the Z language from the viewpoint of users and tool makers. These include lexical matters, the representation of Z in ASCII symbols, overall questions of syntax and usage, detailed areas of syntactic divergence and some unresolved issues in the type-checking system. [illegible] Programming languages [illegible]

The syntax and lexis of the specification language Z

CONTENTS

1 Introduction

The philosophy in the development of the specification language Z has been to establish the requirements for the language by carrying out case studies to elucidate what features are beneficial in practice and can reasonably be included in the language without prejudice to the overall aims of simplicity and the ability to provide a formal description of the language. Naturally, during the course of this development various divergences of language features emerged, as exemplified in the differing styles found in the collection of case studies [Hayes 1987] which formed the first widely available description of Z. These divergences should not be exaggerated: they amount to no more than are found in differing implementations of a language such as Algol60, so there is no doubt that the language exists as a recognisable (and useful) entity. However, as the language becomes used in an industrial setting, where the acceptability of specifications needs to be established in detail, and as tools are developed to manipulate the language, it becomes more and more important to standardise a language definition. This requires the development of documents setting out the presentation of the language and its semantics, with particular application to proof obligations and the logic to be used for reasoning about specifications.

The presentation of the language has been addressed in a Programming Research Group report prepared by King *et al* [1989], which gives concrete and abstract syntaxes for the language, and in a Z reference manual published by Spivey [1989]. These two references are referred to as K and S in what follows. The semantics of Z were described in Spivey's thesis and published in [Spivey 1988] which also contained a concrete syntax for the core subset of Z which was the subject of the semantic definition. This book is of fundamental importance in the definition of Z, but it does not settle all issues. On questions of presentation, both of Spivey's books deal with smaller languages than that described in the PRG report and even within the subset common to both there are divergences. On questions of semantics, proof obligations and a proof theory for Z still remain undefined.

This memorandum is based on experience in the use of Z at RSRE (see [Sennett 1987], [Randell 1990]) and is presented as a contribution to the standardisation of Z. It deals mainly with the presentation issues which need to be settled in the development of a Z standard from the viewpoint of users and tool makers. These are lexical matters, the representation of Z in ASCII symbols, overall questions of syntax and usage, detailed areas of syntactic divergence and some unresolved issues in the type-checking system.

2 Lexical matters

Because of the non-standard character sets and the use of boxes to delimit scope, lexical analysis for Z is unusual, although it has not been extensively discussed. In many cases the actual presentation of the Z text on a VDU or printed on paper will be controlled by data structures which are machine specific and inappropriate to standardise. However, some rules affect the user interface and ought to be made explicit. It is also important to be able to transfer a specification from one machine to another with the minimum of fuss and for this simple, lexical conventions are necessary. Topics which need addressing are the termination of identifiers, decoration, the significance of new lines, the use of superscripts and a standard for Z text which is capable of representation in the graphic ASCII characters.

Identifiers

The termination of identifiers is important because they are frequently juxtaposed. Should $x+y$ be treated as three identifiers or one? Is $\alpha\beta$ two identifiers or one? Are Δ and Ξ reserved symbols? Are λ, μ and θ reserved symbols or can they be used in identifiers? These questions are not addressed in any of the references, so the following scheme is proposed.

1. The following symbols are reserved:

$$\theta\ \lambda\ \mu\ \langle\ \rangle\ \triangleq\ !\ "\ \$\ '\ (\ )\ ,\ ;\ ?\ [\ ]\ \{\ |\ \}\ \vdash\ \bullet\ \forall\ \exists\ \mathbb{P}\ \langle\!\langle\ \rangle\!\rangle$$

2. Alphanumeric identifiers may be formed from the characters A to Z, a to z, 0 to 9, the Greek letters not in the reserved symbols and underline. An identifier may not start with a numeral or underline or end with an underline. Any other character or a space, a new line, a subscript or a superscript terminates an alphanumeric identifier.

3. A symbolic character is any graphic character other than the ones given above. A symbolic identifier is a sequence of symbolic characters and is terminated by any non-symbolic character or by a space or layout as for an alphanumeric identifier.

This scheme is simple to implement and seems to cause few problems. The choice of reserved symbols is somewhat arbitrary but is done on the basis that major syntactic indicators should not be used in identifiers as this makes for difficult reading.

Decoration

The standard Z decorations are *!*, *?*, *'* and the subscripts. Left unspecified is whether more than one decoration of the same type can be applied. For example, is *a!!* an allowable identifier? Furthermore, is the order of decorations important? Is $a_1!$ different from $a!_1$? Because of the semantics attached to the decorations it seems that one can only argue for more than one prime to be applied, and even this argument is somewhat dubious. For reasons of simplicity it seems best to make the rule that only one subscript and only one decoration from the set *?*, *!*, *'* may be applied and that the order of application of the two types of decoration is immaterial.

A separate issue is that S requires subscripts to be digits whereas K allows any sequence of characters. The K approach is desirable although it leads to the question of whether spaces are significant or not within a subscript.

New lines

Predicates frequently extend over more than one line and yet Z has the convention that vertically stacked predicates are to be treated as conjoined. This latter convention is useful because it introduces a weakly bound "and" operator which is often convenient. S suggests that new lines may be ignored before or after certain characters whereas K suggests that the "stacking" new line should be a separate character. The K approach has the merit of simplicity and is likely to lead to less puzzling error messages as well as making explicit the structure introduced.

Superscripts

Superscripts have proved to be particularly tedious for simple lexical analysis. The problem is that iteration is represented by using superscripts and some post fixed operators, such as transitive closure, are written in the superscript position. An expression of the form A^+ could therefore represent the application of either the closure operator or iteration in the pathological case of the identifier + being defined to be an integer. From the lexical point of view, A^+ is made up of two identifiers in the first case (A and the closure operator) and three in the second (A, the superscript representing iteration and +). As a compromise, a useful rule which has been adopted is to treat superscripts which consist solely of symbolic identifiers as a post fixed operator and all others as iteration.

ASCII representation

Transfer of Z texts between machines is a surprisingly frequent occurence. Experience shows that such transfers are best carried out within the set of 96 printing characters which can be transferred with the least likelihood of problems. For this, it is necessary to define representations for Z boxes and the set of mathematical characters which enables the standard Z library operators and functions to be used. The obvious technique is to represent each of these characters by a mnemonic, preceded by an escape character. A proposed set of mnemonics is as follows, based on that in K.

Mnemonic	Meaning
Z	Start of Z
EZ	End of Z
SB	Schema box
ESB	End schema box
ST	Such that
TH	Start of theorem
ETH	End of theorem
SR	Start vertical rule
ER	End vertical rule
UD	Unique definition (double rule)
SW	Start where
EW	End where
where	where
SI	Start indent
EI	End indent
UP	Start superscript or end subscript
DOWN	End superscript or start subscript

This differs from the list in K in the use of UD rather than GE for the double rule indicator, delimiters for *where*-phrases and in the indications of the start and end of subscripts and superscripts. UD has been chosen rather than GE simply because this seems to be the more important information conveyed by the double rule and also because the double rule is used for non-generic definitions. The *where*-phrase is the only place in Z where use before declaration is necessary: delimiters make it possible to change the order of the declarations with respect to the predicates during lexical analysis, which simplifies the production of tools. The start and end of subscripts, superscripts and versions is indicated by six separate mnemonics in K, which is rather complicated and so UP and DOWN have been used instead.

The use of these mnemonics is indicated by the following syntax, in the notation of BS6154 [BSI 1981].

file = text, {'Z', zphrase, 'EZ', [text]};

The *file* is the total Z specification, while a *text* is the narrative separating fragments of the formal text, a *zphrase*. A *text* is any sequence of characters excluding the escape character.

zphrase = schema | theorem | unique_def | ztext;

At the top level of the lexical structure, a formal text is either a schema, a theorem, a unique definition or is some other form of Z text (*ztext*) such as definitions or global constraints.

schema = ztext, 'SB', ztext, ['ST', ztext], 'ESB';

The first *ztext* is the schema name, the second the signature and the optional third, the predicate.

theorem = 'TH', ztext, 'ETH';

A theorem is simply formal text delimited by the appropriate mnemonics. A unique definition is distinguished by the double rule:

unique_def = 'UD', ztext, ['ST', ztext], 'ER';

A *ztext* contains either character strings (*zchar*) or delimited, formally structured, text (*zpair*).

ztext = {zchar | zpair};

A *zchar* is made up of any sequence of characters, including escape sequences other than those given above. These additional escape sequences are used to represent the mathematical symbols. A *zpair* contains Z text delimited as follows:

```
zpair = 'UP', ztext, 'DOWN'
      | 'DOWN', ztext, 'UP'
      | 'SW', ztext, 'where', ztext, 'EW'
      | 'SR', ztext, ['ST', ztext], 'ER'
      | 'SI', ztext, 'EI'
```

The Z structure beginning *SR* is introduced at this level in the syntax to allow it to be used in *where*-phrases. The additional escape sequences used to represent the mathematical symbols are given on the next page.

Mnemonic	Graphic	Meaning
def	≙	Schema definition
lang	⟪	Left angle bracket for disjoint union
rang	⟫	Right angle bracket or piping
not	¬	Not
and	∧	And
or	∨	Or
imp	⇒	Implication
iff	⇔	Equivalence
all	∀	For all
exi	∃	There exists
spot	•	Then
mem	∈	set membership
prod	×	Cartesian product
pset	ℙ	Power set
fcomp	⨾	Forward composition
over	⊕	Overriding
neq	≠	Not equals
int	∩	Intersection
uni	∪	Union
subs	⊆	Subset
psubs	⊂	Proper subset
nem	∉	Not member
dint	⋂	Distributed intersection
duni	⋃	Distributed union
fset	𝔽	Finite set
null	∅	Empty set
rel	↔	Relation
comp	∘	Composition
dres	◁	Domain restriction
dsub	⩤	Domain subtraction
rres	▷	Range restriction
rsub	⩥	Range subtraction
limg	⦇	Left image bracket
rimg	⦈	Right image bracket
inv	$^{-1}$	Inverse
map	↦	Maplet
Nat	ℕ	Natural numbers
geq	≥	Greater than or equal
leq	≤	Less than or equal
cat	⁀	Concatenation
ires	↿	Index restriction
sres	↾	Sequence filter
lseq	⟨	Left sequence bracket
rseq	⟩	Right sequence bracket
thm	⊢	Turnstile
pfun	⇸	Partial function
tfun	→	Total function
ffun	⇻	Finite function
pinj	⤔	Partial injection
tinj	↣	Total injection
finj	⤕	Finite injection
psur	⤀	Partial surjection
tsur	↠	Total surjection
bij	⤖	Bijection

In addition it is useful to have mnemonics for the Greek characters as follows:

Mnemonic	Graphic	Mnemonic	Graphic
Ga	α	GC	Γ
Gb	β	GD	Δ
Gc	γ	GF	Φ
Gd	δ	GH	Θ
Ge	ε	GL	Λ
Gf	φ	GP	Π
Gg	χ	GS	Σ
Gh	θ	GQ	Ψ
Gi	ι	GW	Ω
Gk	κ	GX	Ξ
Gl	λ		
Gm	μ		
Gn	ν		
Gp	π		
Gq	ψ		
Gr	ρ		
Gs	σ		
Gt	τ		
Gu	υ		
Gw	ω		
Gx	ξ		
Gy	η		
Gz	ζ		

3 General issues of syntax

The writing down of the syntax for a language is never a cut and dried matter. In separating out the question of lexical analysis, for example, a decision has been taken not to express syntactically things which are best left specified algorithmically. A similar case could be made for not giving priorities of the operator symbols in the syntax as this results in a much more compact syntax and is the treatment followed by both S and K. The question of priorities is discussed later. On the whole from the tool writing point of view, it is probably better to specify binding powers in the syntax as this is more precise.

At the other extreme one should not try to express semantic issues syntactically as this tends to result in ambiguous grammars. This again is a case in point for Z because both sources of Z syntax are ambiguous in the treatment of terms and predicates: a schema reference for example may either be a term or a predicate and it is only the semantic context which determines which it is. Nevertheless, from the didactic point of view, it is useful to express the syntax in this way and an official standard for Z could well include two syntaxes, one for explanatory purposes and one for tool writers, the latter being unambiguous.

A similar problem is that a syntax may not allow parsing on the basis of testing one symbol. The tool writer obviously prefers a syntax to be one-trackable in this way, but this should not be at the cost of rebarbative extra symbols and syntax. For example, the syntax for explicit sets and set comprehensions suffers from this problem. The sequence $\{a, b, c....$ could be the start of an explicit set $\{a, b, c\}$ or a comprehension $\{a, b, c : T...\}$. This does not seem to cause sufficient trouble to warrant a change to the syntax as it is possible to use a small amount of look-ahead to resolve the problem and still use a one-track syntax analyser. Generic parameters and instantiation on the other hand do seem to call for a change to the syntax as it is possible to follow fairly widespread practice. In this case the problem arises with a sequence like $A[X, Y, ...$ which could either start a generic definition or be the instantiation of a generic variable in a global constraint. In this case one can solve the problem by requiring instantiations to be in the suffix position, as in $A_{[X, Y..}$ which is common practice among Z users anyway.

A similar case occurs in μ-constructions, where it is sometimes desirable to omit the term following the spot. This is allowed in S although at the cost of requiring λ and μ terms to be enclosed in brackets. Omitting the term following the spot without this delimiter leads to many difficulties. For example, in the term $\lambda\ a : \mu\ b : \mathbb{P}\ T \bullet b \bullet a$, the μ-term cannot be compiled until the result of the λ-term has been found and a simple look-ahead will not resolve the matter. Also, a term such as $\mu\ a : \mu\ b : \mathbb{P}\ T \bullet a$ is ambiguous. The simpler approach of always requiring the spot and its following term seems preferable.

In summary, the syntax given in the appendix resolves these issues as follows:

1. The term and predicate ambiguity is resolved by allowing an atomic term to be a bracketted predicate and allowing one of the productions of the rule for a predicate to be a term.

2. The distinction between explicit sets and set comprehensions is made by means of an artificial terminal symbol (called *explicit_set*) which is inserted into the output of the lexical analyser by means of a simple look-ahead procedure.

3. An instantiation is introduced by an opening square bracket in the suffix position.

4. The spot may not be omitted from λ and μ terms.

4 Atomic expressions

This section lists points of divergence concerning the use of identifiers and constants in expressions. As the issues are fairly straightforward they are simply listed with a few comments.

Declaration before use

An identifier may be used before it is defined in K but not in S. Declaration before use is clearly preferable as it enforces a good style of presentation and eases the production of tools.

Instantiation

The type instance of a generic term in an expression is usually inferred from the context, but it may be supplied explicitly. In most cases, a term is generic in one type only: for example the union operator acts between sets of the same type and also delivers a set of this type. However, generic terms can depend on more than one type: for example the *disjoint* operator depends on the type for the indexing set as well as the type of the family of sets which are disjoint. In this case two types may have to be supplied and the question is how is the correspondence between the generic type and its instantiating term to be shown. S allows one method, namely the terms are supplied in the order in which the generic types were introduced in the definition of the generic term: K allows this method but also a method in which the generic terms are explicitly named. The two methods are shown below:

$$disjoint_{[\mathbb{N},\, Char]} \qquad disjoint_{[I = \mathbb{N},\, X = Char]}$$

This is one of those bicycle shed issues which can occupy standards committees for whole mornings. Probably the best thing is to bow to the inevitable and accept both as in K.

A related issue is that schema renaming is allowed in K but not in S. This a facility for renaming identifiers in a schema, so that it may be used in schema composition, for example. This is allowed in K in explicit schema definitions, for example

$$A \mathrel{\widehat{=}} B \mathbin{;} C_{[b\,/\,a]}$$

which renames the schema identifier *a* in schema *C* to be *b*. In view of the dependence of Z on the use of identifiers in schema inclusions and schema expressions, this seems to be a useful facility. A purist would argue that it is a superfluous complication and that specifications should be developed with coherent naming schemes. This is a tenable position and the repeated use of schema renaming can be confusing. However, before abandoning this facility completely it would be worthwhile investigating its use in practical situations.

Free standing operators

The notation for the set which corresponds to a function or relation, as opposed to the result of applying the function or relation is not altogether fixed. In S, the notation is indicated by supplying the underline as a placeholder for the missing parameters, for example *(_ + _)*, but the syntax for this is not given explicitly. Further, this notation is naturally extended to constructions like *(_ + y)* for the function which adds *y* to its argument. With this extension, the brackets round the operator become essential to establish the binding.

Some atomic forms

Denotations for characters and sequences of characters are provided in K but not in S. These are essential for error messages. Denotations for the predicates *true* and *false* are provided in S but not in K. These are useful. Explicit bags are provided in S, but not in K. Explicit bags are rarely useful, except as examples. Having them in the syntax requires the symbols ⟦ ⟧ to be reserved which is a complication and makes two useful symbols unavailable.

The *where* phrase

This very useful construction is missing in S, but present in K. The semantics of this phrase are easily provided in terms of existential quantifiers and its use adds considerably to the readability of Z specifications, so there seems to be no compelling reason to omit it. The signature in a *where* phrase is usually provided in the form of an axiomatic definition, but a syntactic definition should also be allowed as this has even simpler semantics.

Conditional constructions

An *if then else* construction is omitted from both S and K. This is another device which improves the readability of specifications as the equivalent logical expression is both cumbersome and repetitive. The only barrier to its adoption is a proposal for concrete syntax which would fit in with the Z style.

5 Operator precedence

Precedence in computer languages has always presented problems. In Algol60, user-defined operators were not allowed so it was possible to give the precedence of all the operators in the language explicitly. In Algol68 and ML, user defined operators are allowed and the precedence of the operator symbols can be supplied as an integer in a priority declaration within the language. In APL, user defined operators are also allowed, but all operators have the same precedence. All of these approaches suffer from disadvantages in one way or another. Having no precedence leads to a rapid multiplication of brackets which is unsightly, confusing and quite unacceptable. Having user definition of precedence on the other hand is just as confusing. In practice one quickly forgets the priority settings for all but the standard operators and a long expression becomes hard for the reader to parse. On grounds of portability, either precedence should be fixed, or the priority definition should be part of the language. On grounds of simplicity the former is preferred.

Similar comments apply to the syntactic status of the operator symbols, which indicates whether they are infix, prefix or postfix in relation to their arguments. In the case of Z, it is also necessary to indicate whether the operator is a generic set or not and whether an operator is a relation or a function. There is some divergence as to possibilities here, as indicated in the table below:

Status	Examples	Provided in S	Provided in K
Operators			
Infix	_ * _	Yes	Yes
Prefix	*hd* _	No	Yes
Postfix	$\_^{-1}$	Yes	Yes
Distprefix	⟦_⟧_	Yes	Yes
Distpostfix	_⟦_⟧	No	Yes
Enclosed	⟦_⟧	No	Yes
Generic sets			
Infix	_ → _	Yes	Yes
Prefix	*seq* _	Yes	Yes
Postfix	_ *list*	No	Yes
Relations			
Prefix	*ordered* _	Yes	No
Infix	_ > _	Yes	Yes

Prefix operators are not needed in S because their place is taken by simple function application. They are provided in K which allows them to be given a different precedence and binding from function application. The enclosed form of an operator is not provided by S. Its most frequent use is in the meaning function used in denotational semantics. Postfixed generic sets are not provided in S: they are not common in Z, but would enable some types to be expressed in a VDM style. Prefix relations are not provided in S, because they are treated as a semantic variation on function application (as determined by the types). However, a separate syntactic status would also enable this case to be given a different precedence from function application. The syntax given in the appendix follows K in this question, although there seems to be no reason to exclude any of the possibilities.

S provides a priority and syntactic status for the operators in the Z library, but no mechanism for introducing new operator symbols or assigning their priorities: this is supposed to be supplied by some mechanism outside the language. K also provides no mechanism for supplying either precedence or syntactic status. This is not satisfactory, but as far as precedence is concerned, the best approach would seem to be to have all infixed operators having the same precedence, thus eliminating the need for a priority definition mechanism. Specifications do not frequently contain extensive arithmetic expressions, which is the only case where precedence of the operators (* over +, for example) can be assumed. Z already contains a very extensive precedence structure, as indicated in the table below, and to put an extra layer of precedence into operator expressions would seem to impose too much of a burden on the reader.

Order of increasing precedence for predicates

; or stacked predicates
Quantified expressions
The logical connectives in the order $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$
$=, \in$ and relations

Order of increasing precedence for terms

Cartesian product
λ and μ terms
infix generic sets
infix operators
function application
prefix operators and generic sets
postfix operators and generic sets
selection

Given a fixed precedence, the only problem is a mechanism for defining the syntactic status of identifiers, which may be achieved by the use of underline as a place holder for the parameters. In order to make this defining role clear, the syntax should specify that an identifier is required, rather than the appropriate variant of an identifier. (Thus the syntax rule should be in the form *declaration_name = '_', identifier, '_'*, not *declaration_name = '_', infix, '_'*.) It is also necessary to specify the difference between an infix function and a relation. This is allowed for in K by enclosing the identifier and its placeholders in brackets to indicate an infixed operator, as opposed to an infixed relation.

The question of binding is related to the question of precedence. In K, *all* the connectives bind on the left: in S $\Rightarrow$ binds on the right, following common practice among logicians. In both, infix generic sets bind to the right as the most common ones are the functions. In K, prefix operators and generic sets bind on the right. As a precedence has been given for the logical connectives, it seems reasonable to follow common practice and make $\Rightarrow$ bind on the right, as in S. The other conventions are also very reasonable, although the ability of the user to define infixed generic sets may lead to some surprises.

One final minor point is that because of the low precedence of the Cartesian product operator, a Cartesian product is always required to be enclosed in brackets. This view is adopted by K, but not by S.

6 Definitions

Re-declaration

It should not be possible for an identifier to be declared twice within the same scope. Because of the possibility of schema inclusion, the issue is not quite straightforward in Z. It is clearly desirable to be able to include schemas which have part of their signature in common, although as a question of style, a schema whose signature is totally contained within the current scope should be treated as a predicate, rather than an inclusion. However a *declaration* introducing an identifier which is already defined within the current scope should be treated as an error. This is because it is likely to be a mistake and if intentional, the only effect is to introduce an extra constraint, which more properly belongs to the predicate part of whatever is being defined. Neither S nor K make a clear statement on this issue, although it is certainly the case that if an identifier is declared twice, by either inclusion or declaration, the types should be compatible. This would seem to rule out a style which was used in the early days of Z, of re-defining a schema with the same name, but a gradually increasing signature, obtained by including the old schema and adding declarations.

Clearly it should be possible to re-define identifiers within a different scope. There is a minor problem here in that it is difficult to re-define an identifier with a syntactic status other than ordinary identifier. This is because the syntax only allows for identifiers in declarations and previously defined identifiers will have the syntactic status of their previous definition. For example, a library identifier such as *seq* is already defined as a generic prefixed set and will be recognised by the lexical analyser as such and consequently may not appear in a definition.

Horizontal schemas

The use of the horizontal form for a schema is allowed in both S and K, but as it can only be used within a schema definition it is a rather pointless complication. The use of ordinary square brackets to delimit it, causes the syntax to depart from a one track form: this character also delimits generic parameters and instantiations, so it is being rather overworked. This is recognised in K which uses special brackets but this also is a complication.

Implicit declaration of schemas

The use of schema names beginning Δ or Ξ is a well-known convention in Z and the question arises as to whether such a schema can be used without previous declaration. Clearly, it should be possible to define schemas having these names for special purposes, but when they have their conventional meaning there seems little point in requiring them to be defined. If this view is adopted, then once a schema has been defined implicitly, it should not be possible to re-define it.

Allowable terms for use in declarations

S, on page 54, introduces some restrictions on terms which may be used to the right of a colon in a declaration. Clearly, $T : \mathbb{P}\, T$ is nonsensical, so these terms may not refer to the term being defined. This also applies to syntactic definitions. However a construction like $T : \mathbb{P}$ *some set*; $f : T \rightarrow T$ is both useful and a fairly common occurrence: it is not clear whether Spivey's remarks are meant to exclude this case, and if so, why.

Generic datatype definitions

Generic datatypes, for example trees of generic type, occur in Z *argot* but not in S or K. They are clearly useful and should not present semantic difficulties, so there seems to be no good reason to exclude them.

Characteristic tuples

The characteristic tuple of a series of declarations is used to give the type for some of the Z constructions: for example, the set $\{A; B\}$ where A and B are schemas has the type $\mathbb{P}(A \times B)$. This presents a problem for the λ constructions, or rather the *application* of a λ term in the case where the schemas have overlapping signatures. For example, consider the following definitions:

```
┌─ A ────┐  ┌─ B ────┐  ┌─ a ────┐  ┌─ b ────┐
│ x, y : ℕ │  │ y, z : ℕ │  │ A      │  │ B      │
└────────┘  └────────┘  ├────────  ├────────
                        │ y = 1  │  │ y = 2  │
                        └────────┘  └────────┘
```

$f == \lambda A;\ B \bullet x + y + z \qquad f(a, b) = 2$

The problem with the last constraint is not with whether the specification has a model, but rather with whether the statement has a meaning. Clearly it does not, but this can only be established semantically and this ought not to be the case with a well formed language. The simplest restriction is to forbid overlapping signatures in λ terms.

Documents

A serious use of Z necessarily involves specifications in several parts. K presents a scheme of documents for this whereas S does not consider the issue. Module schemes are a difficult enough issue for ordinary implementation languages and the implications for specification languages have not been thoroughly studied so one can have sympathy with this point of view. However, the K scheme seems to be perfectly workable and should be adopted as an interim measure. In our own work on tools it has also been found useful to provide an export statement which defines which identifiers may be exported to other documents. This simply follows good practice in similar implementation language module systems.

Theorems

A notation for theorems is given in K, but not in S. This should be present, although again one can see the need for future study of the impact of mechanical theorem proving aids. Ideally, the language should allow the outlines of a proof to be written down and how this is done may be affected by the nature of the tools used. However, K again gives a workable notation which is useful in the interim.

7 Types

The type system in Z is a useful and elegant compromise between the granularity of the typing system and the ability to detect ill-formed specifications. Types in Z are built up from the given sets and types constructed from these using the powerset, tuple and schematype constructors. Schema types are formed from a mapping of the identifiers of the schema signature to their corresponding types and herein lies the main difficulty with the Z type system. This arises from the ability to decorate a schema, an operation which intuitively should not change the type, but which clearly alters the identifiers. Thus it is desirable, for example, that the set $\{S; S'\}$ should be a *homogenous* relation on S and this would not be so if decoration altered the type.

The rule adopted by both S and K is that the decoration operation, applied to a schema, does not alter its type. This has a number of unfortunate consequences. For example, in the context of the following definitions:

```
┌─ S ──────    ┌─ S₁ ─────    S₂ ≙ S'
│ x : ℕ        │ x' : ℕ
└──────────    └──────────
```

S_1 does not have the same type as either S or S'. Moreover, the decoration has to be "frozen" into the type at the time of a schema definition (this is because a more complicated schema term could introduce further elements into the signature). Consequently, S_2 does not have the same type as S'. This is clearly unacceptable.

The solution originally proposed by Spivey, of having schematypes depend on the decorated identifiers, but using equality modulo a decoration for equality of types, seems to be workable and to avoid these difficulties.

Appendix - a version of the Z Syntax

This syntax is given in British Standard form (BS 6154:1981) and has the following features:

1) Denotations of the terminal symbols are enclosed in quotes.
2) [and] indicate optional symbols.
3) { and } indicate repetition, that is, a possibly empty sequence of symbols.
4) Each rule has an explicit final character (a semi-colon).
5) Brackets group items together.
6) (* and *) indicate a comment.
7) A comma is the concatenate symbol, an equals sign the defining symbol and a vertical line the alternate symbol.

Named terminal symbols

id	standard identifier, including decoration
document	a specification module
decor	?, !, ', or subscript
EZ	end of Z
NL	hard new line
SI	start indentation
EI	end indentation
finish	end of file
SR	start vertical rule
ER	end vertical rule
UD	unique (generic) definition
inset	infixed generic sets
preset	prefixed generic sets
postset	postfixed generic sets
op	infix operator
encop	lhs of enclosed operator
distinop	lhs of distributed infix operator
distpreop	lhs of distributed prefix operator
eop	delimiter of two part operators
preop	prefix operator
postop	postfix operator
sconst	numbers and the constants
TH	start theorem
ETH	end theorem
SW	start of where phrase
where	where phrase delimiter
EW	end of where phrase
rel	relational operator
explicit_set	{ when indicating start of set display
pre	pre-condition schema operator
SB	start schema box (after name)
ST	middle line of schema box
ESB	end schema box

Rules

(* For this syntax, narrative text up to and including the Z delimiter is assumed to be disposed of by the lexical analyser *)

```
z_text = [z_phrase], finish
       | z_phrase, z_sep, z_text;

z_sep = list_sep | EZ;
```

```
list_sep = ';'  | NL;

z_phrase = given_set_def
           | definition
           | constraint
           | theorem
           | import
           | export;

given_set_def = '[' , given_ids, ']' ;

given_ids = id, { ',' , id };

definition = axiomatic_def
             | syntactic_def
             | datatype_def
             | schema_def;

constraint = pred;

theorem = '⊢' , pred
          | TH , [gen_params], [hyps], '⊢' , pred_list, ETH;

hyps = hyp, {list_sep, hyp};

hyp = pred | dec | pred,  '|' , dec;

import = doc, {doc};

doc = document, [ decor ], [instantiation];

export = id, 'keep', idslist;

ids = id | inset | preset | postset | op | rel | encop, eop
    | distinop , eop | distpreop , eop | preop | postop;

idslist = ids, { ',' , ids };

reference = ( id | id , '$' , id ), [instantiation];

instantiation = '[' , inst_list, ']' ;

inst_list = inst_term_list | binding_list | rename_list;

(* The two forms of instantiation and schema renaming are all treated as instantiation in
this syntax: the various possibilities are distinguished semantically *)

inst_term_list = term, { ',' , term };

binding_list = id, '=' , term, { ',' , id, '=' , term };

rename_list = id, '/', id, {',', id, '/', id };

axiomatic_def = liberal_def
                | unique_def
                | generic_def;

liberal_def = dec, [ '|' , pred ]
              | SR, def_body, ER;
```

```
def_body = dec_list, [ ST , pred_list ];

unique_def = UD, def_body, ER;

generic_def = decl_name, gen_params, ':' , term, '|' , pred
              | UD, gen_params, def_body, ER ;

gen_params = '[' , given_ids, ']' ;

dec_list = dec, {list_sep, dec};

dec = decl_name, {list_sep, decl_name}, ':' , term;

(* decl_name has options to indicate the syntactic status of the operator being defined *)

decl_name = id | id , '_' | '_' , id | id , '_' , id
            | '(' , '_' , id, '_' , ')'
            | '_' , id , '_'
            | id, '_' , id , '_'
            | '_' , id, '_' , id;

syntactic_def = syn_def_id
                | decl_name, gen_params, '==' , term
                | UD, gen_params, syn_def_list, ER;

syn_def_id = decl_name, '==' , term;

syn_def_list = syn_def, {list_sep, syn_def};

syn_def = syn_def_id | syn_def_ids;

(* syn_def_ids defines prefix, postfix and infix generic sets *)

syn_def_ids = id, id, '==' , term
              | id, id, id, '==' , term;

datatype_def = id, '::=' , branch, { '|' , branch };

branch = id | id, '«' , term, '»' ;

schema_def = id, [gen_params], schema_definition;

schema_definition = '≙', schema_term | schema;

schema = SB, local_dec_list, [ ST, pred_list ], ESB;

pred_list = pred, {list_sep, pred};

local_dec_list = (dec | inclusion), [list_sep, local_dec_list];

inclusion = ( id | id, '$' , id), [instantiation];
(* Only schema renaming is allowed here *)

explicit_constr = tuple
                  | explicit_set, '}'
                  | explicit_set, termlist1, '}'
                  | '⟨' , [termlist1], '⟩' ;

termlist1 = term, { ',' , term };
```

```
tuple = '(' , term, ',' , termlist2, ')'
      | 'θ' , reference;

termlist2 = term, { ',' , term };

(* aform = atomicformulae, and includes predicates. The distinction between terms and
predicate is checked semantically *)

aform = 'ℤ' | 'Char' | sconst
      | reference
      | aform, '.' , id
      | '(' , product, ')'
      | explicit_constr
      | '{' , local_dec_list, [ '|' , pred ], [ '•' , term ], '}'
      | '(' , partials, ')'
      | encop, term, eop
      | SW, ax_dec_list, where, pred_list, EW
      | SW, syn_def_list, where, pred_list, EW
      | '(' , pred, ')' ;

product = term, '×', term, { '×', term };

ax_dec_list = local_dec_list, '|' , pred
            | SR, dec_list, ST, pred_list, ER;

(* partials corresponds to various forms of partial application *)

partials = '_' , rel, '_'
         | '_' , op, form2
         | aform, op, '_'
         | '_' , op, '_'
         | '_' , distinop, term, eop
         | aform, distinop, '_' , eop
         | '_' , distinop, '_' , eop
         | distpreop, '_' , eop, '_'
         | distpreop, term, eop, '_'
         | distpreop, '_' , eop, form3
         | encop, '_' , eop
         | preop, '_'
         | '_' , postop;

(* The following syntax rules define the priority of the various operator symbols. The
weakest binding is the infixed generic sets such as → *)

formula = form1, {inset, form1};

(* infixed operators *)

form1 = [ form1, op ], form2;

(* function application *)

form2 = [form2], form3;

(* prefix function application and generic sets *)

form3 = preop, form3
      | preset, form3
      | distpreop, term, eop, form3
```

```
           | 'ℙ', form3
           | form4;

(* postfix function application and generic sets *)

form4 = form4, distinop, term, eop
           | form4, postop
           | form4, postset
           | aform;

comprehension = 'λ' , local_dec_list, lambda_set
                         | 'μ' , local_dec_list, lambda_set;

lambda_set = [ '|' , pred ], '•' , term;

term = comprehension | formula;

apred = SI , pred_list, EI | term;

rel_exp = term, '∈ ', term
             | term, '= ', term, [tail]
             | term, rel, term, [tail]
             | apred;

tail = rel, term, [tail]
     | '= ', term, [tail];

(* priority of connectives: *)

log_exp = log_exp1 | log_exp, '⇔', log_exp1;

log_exp1 = log_exp2 | log_exp1, '⇒', log_exp2;

log_exp2 = log_exp3 | log_exp2, '∨', log_exp3;

log_exp3 = log_exp4 | log_exp3, '∧', log_exp4;

log_exp4 = { '¬' }, rel_exp;

quant_exp = quant, dec_list, [ '|', pred ], '•', pred;

quant = '∃', '∃1', '∀' ;

pred = quant_exp | log_exp;

schema_term = quant_sexp | log_sexp;

quant_sexp = quant, dec_list, '•', schema_term;

log_sexp = log_sexp1 | log_sexp, '⇔', log_sexp1;

log_sexp1 = log_sexp2 | log_sexp2, '⇒', log_sexp1;

log_sexp2 = log_sexp3 | log_sexp2, '∨', log_sexp3;

log_sexp3 = log_sexp4 | log_sexp3, '∧', log_sexp4;

log_sexp4 = spec_sexp | '¬', log_sexp4;
```

```
spec_sexp = spec_sexp, '/', '(' , id_list, ')'
          | spec_sexp, '/', reference
          | spec_sexp, 'op', spec_sexp1
          | spec_sexp, '»', spec_sexp1
          | spec_sexp1;

id_list = id, {',', id};

spec_sexp1 = [ 'pre' ], spec_sexp2;

rename = '[' , rename_list, ']' | decor;

spec_sexp2 = '(' , schema_term, ')', [rename]
           | reference
           | schema;
```

References

BSI (1981). Method of defining syntactic metalanguage, British Standards Institution, BS 6154:1981

Hayes I, (1987). Specification case studies, Prentice Hall International series in Computer Science, 1987.

King S, Sorensen I H, Woodcock J, (1987). Z: grammar and concrete and abstract syntaxes, Programming Research Group, University of Oxford.

Randell, G P (1990). Zadok user guide. RSRE Memorandum 4356.

Sennett, C T (1987). Review of the type checking and scope rules of the specification language Z, RSRE report 87017.

Spivey, J M (1988). Understanding Z: a specification language and its formal semantics. Cambridge University Press.

Spivey, J M (1989). The Z notation: a reference manual. Prentice Hall International series in Computer Science.

REPORT DOCUMENTATION PAGE

DRIC Reference Number (if known) ....

Overall security classification of sheet ....Unclassified..

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the field concerned must be marked to indicate the classification eg (R), (C) or (S).

Originators Reference/Report No. MEMO 4367	Month FEBRUARY	Year 1990

Originators Name and Location
RSRE, St Andrews Road
Malvern, Worcs WR14 3PS

Monitoring Agency Name and Location

Title

SYNTAX AND LEXIS OF THE SPECIFICATION LANGUAGE Z

Report Security Classification Unclassified	Title Classification (U, R, C or S) U

Foreign Language Title (in the case of translations)

Conference Details

Agency Reference	Contract Number and Period
Project Number	Other References

Authors Sennett, C T	Pagination and Ref 22

Abstract

This memorandum is presented as a contribution to the standardisation of the specification language Z. It deals with the presentation issues which need to be settled in the definition of the Z language from the viewpoint of users and tool makers. These include lexical matters, the representation of Z in ASCII symbols, overall questions of syntax and usage, detailed areas of syntactic divergence and some unresolved issues in the type-checking system.

Abstract Classification (U,R,C or S)
U

Descriptors

Distribution Statement (Enter any limitations on the distribution of the document)

Unlimited

580/48